Geometry3D

Release 0.1.0

Minghao Gou

Apr 30, 2020

CONTENTS

CHAPTER

ONE

ABOUT GEOMETRY3D

Geometry3D is a simple python computational geographics library written in python. This library focuses on the functions and lacks efficiency which might be improved in future version.

1.1 Core Features

- Basic 3D Geometries: Point, Line, Plane, Segment, Convex Polygen and Convex Polyhedron.
- Basic Attributes Of Geometries: length, area, volume.
- Basic Relationships And Operations Between Geometries: move, angle, parallel, orthogonal, intersection.
- Overload Build-In Functions Such As *__contains__*, *__hash__*, *__eq__*, *__neg__*.
- A Naive Renderer Using *matplotlib*.

1.2 Resources

- Documents: https://geometry3d.readthedocs.io/en/latest/
- Code: https://github.com/GouMinghao/Geometry3D

CHAPTER

TWO

INSTALLATION

Note: Tested on Linux and Windows at the moment.

2.1 Prerequisites

It is assumed that you already have Python 3 installed. If you want graphic support, you need to manually install matplotlib.

2.2 System wide installation

You can install Geometry3D via pip:

```
$ pip install Geometry3D
```

Alternatively, you can install Geometry3D from source:

```
$ git clone http://github.com/GouMinghao/Geometry3D
$ cd Geometry3D/
$ sudo pip install .
# Alternative:
$ sudo python setup.py install
```

Note that the Python (or pip) version you use to install Geometry3D must match the version you want to use Geometry3D with.

2.3 Virtualenv installation

Geometry3D can be installed inside a virtualenv just like any other python package, though I suggest the use of virtualenvwrapper.

CHAPTER

THREE

TUTORIALS

3.1 Creating Geometries

3.1.1 Creating Point

Creating a Point using three cordinates:

```
>>> from Geometry3D import *
>>> pa = Point(1,2,3)
>>> pa
Point(1, 2, 3)
```

Creating a Point using a list of coordinates:

```
>>> pb = Point([2,4,3])
>>> pb
Point(2, 4, 3)
```

Specifically, special Point can be created using class function:

```
>>> o = origin()
>>> o
Point(0, 0, 0)
```

3.1.2 Creating Vector

Creating a Vector using three cordinates:

```
>>> from Geometry3D import *
>>> va = Vector(1,2,3)
>>> va
Vector(1, 2, 3)
```

Creating a Vector using two Points:

```
>>> pa = Point(1,2,3)
>>> pb = Point(2,3,1)
>>> vb = Vector(pa,pb)
>>> vb
Vector(1, 1, -2)
```

Creating a Vector using a list of coordinates:

```
>>> vc = Vector([1,2,4])
>>> vc
Vector(1, 2, 4)
```

Specifically, special Vectors can be created using class functions:

```
>>> x_unit_vector()
Vector(1, 0, 0)
>>> y_unit_vector()
Vector(0, 1, 0)
>>> z_unit_vector()
Vector(0, 0, 1)
```

3.1.3 Creating Line

Creating Line using two Points:

```
>>> from Geometry3D import *
>>> pa = Point(1,2,3)
>>> pb = Point(2,3,1)
>>> l = Line(pa,pb)
>>> l
Line(sv=Vector(1, 2, 3),dv=Vector(1, 1, -2))
```

Creating Line using two Vectors:

```
>>> va = Vector(1,2,3)
>>> vb = Vector(-1,-2,-1)
>>> l = Line(va,vb)
>>> l
Line(sv=Vector(1, 2, 3),dv=Vector(-1, -2, -1))
```

Creating Line using a Point and a Vector:

```
Line(sv=Vector(1, 2, 3),dv=Vector(-1, -2, -1))
>>> pa = Point(2,6,-2)
>>> v = Vector(2,0,4)
>>> l = Line(pa,v)
>>> l
Line(sv=Vector(2, 6, -2),dv=Vector(2, 0, 4))
```

Specifically, special Lines can be created using class functions:

```
>>> x_axis()
Line(sv=Vector(0, 0, 0),dv=Vector(1, 0, 0))
>>> y_axis()
Line(sv=Vector(0, 0, 0),dv=Vector(0, 1, 0))
>>> z_axis()
Line(sv=Vector(0, 0, 0),dv=Vector(0, 0, 1))
```

3.1.4 Creating Plane

Creating Plane using three Points:

```
>>> from Geometry3D import *
>>> p1 = origin()
>>> p2 = Point(1,0,0)
>>> p3 = Point(0,1,0)
>>> p = Plane(p1,p2,p3)
>>> p
Plane(Point(0, 0, 0), Vector(0, 0, 1))
```

Creating Plane using a Point and two Vectors:

```
>>> p1 = origin()
>>> v1 = x_unit_vector()
>>> v2 = z_unit_vector()
>>> p = Plane(p1,v1,v2)
>>> p
Plane(Point(0, 0, 0), Vector(0, -1, 0))
```

Creating Plane using a Point and a Vector:

```
>>> p1 = origin()
>>> p = Plane(p1,Vector(1,1,1))
>>> p
Plane(Point(0, 0, 0), Vector(1, 1, 1))
```

Creating Plane using four parameters:

```
# Plane(a, b, c, d):
# Initialise a plane given by the equation
# ax1 + bx2 + cx3 = d (general form).
>>> p = Plane(1,2,3,4)
>>> p
Plane(Point(-1.0, 1.0, 1.0), Vector(1, 2, 3))
```

Specifically, special Planes can be created using class functions:

```
>>> xy_plane()
Plane(Point(0, 0, 0), Vector(0, 0, 1))
>>> yz_plane()
Plane(Point(0, 0, 0), Vector(1, 0, 0))
>>> xz_plane()
Plane(Point(0, 0, 0), Vector(0, 1, 0))
```

3.1.5 Creating Segment

Creating Segment using two Points:

```
>>> from Geometry3D import *
>>> p1 = Point(0,0,2)
>>> p2 = Point(-1,2,0)
>>> s = Segment(p1,p2)
>>> s
Segment(Point(0, 0, 2), Point(-1, 2, 0))
```

Creating Segment using a Point and a Vector:

```
>>> s = Segment(origin(),x_unit_vector())
>>> s
Segment(Point(0, 0, 0), Point(1, 0, 0))
```

3.1.6 Creating ConvexPolygen

Creating ConvexPolygen using a tuple of points:

```
>>> from Geometry3D import *
>>> pa = origin()
>>> pb = Point(1,1,0)
>>> pc = Point(1,0,0)
>>> pd = Point(0,1,0)
>>> cpg = ConvexPolygen((pa,pb,pc,pd))
>>> cpg
ConvexPolygen((Point(0, 0, 0), Point(0, 1, 0), Point(1, 1, 0), Point(1, 0, 0)))
```

Specifically, Parallelogram can be created using one Point and two Vectors:

```
>>> pa = origin()
>>> cpg = Parallelogram(pa,x_unit_vector(),y_unit_vector())
>>> cpg
ConvexPolygen((Point(0, 0, 0), Point(1, 0, 0), Point(1, 1, 0), Point(0, 1, 0)))
```

3.1.7 Creating ConvexPolyhedron

Creating ConvexPolyhedron using a tuple of ConvexPolygens:

```
>>> from Geometry3D import *
>>> a = Point(1,1,1)
>>> b = Point(-1,1,1)
>>> c = Point(-1,-1,1)
>>> d = Point(1,-1,1)
>>> e = Point(1,1,-1)
>>> f = Point(-1,1,-1)
>>> g = Point(-1,-1,-1)
>>> h = Point(1,-1,-1)
>>> cpg0 = ConvexPolygen((a,d,h,e))
>>> cpg1 = ConvexPolygen((a,e,f,b))
>>> cpg2 = ConvexPolygen((c,b,f,g))
>>> cpg3 = ConvexPolygen((c,g,h,d))
>>> cpg4 = ConvexPolygen((a,b,c,d))
>>> cpg5 = ConvexPolygen((e,h,g,f))
>>> cph0 = ConvexPolyhedron((cpg0,cpg1,cpg2,cpg3,cpg4,cpg5))
>>> cph0
ConvexPolyhedron
pyramid set:{Pyramid(ConvexPolygen((Point(1, 1, -1), Point(1, -1, -1), Point(-1, -1, -
↪1), Point(-1, 1, -1))), Point(0.0, 0.0, 0.0)), Pyramid(ConvexPolygen((Point(1, 1,␣
↪1), Point(1, 1, -1), Point(-1, 1, -1), Point(-1, 1, 1))), Point(0.0, 0.0, 0.0)),␣
↪Pyramid(ConvexPolygen((Point(-1, -1, 1), Point(-1, 1, 1), Point(-1, 1, -1), Point(-
↪1, -1, -1))), Point(0.0, 0.0, 0.0)), Pyramid(ConvexPolygen((Point(-1, -1, 1),␣
↪Point(-1, -1, -1), Point(1, -1, -1), Point(1, -1, 1))), Point(0.0, 0.0, 0.0)),␣
↪Pyramid(ConvexPolygen((Point(1, 1, 1), Point(1, -1, 1), Point(1, -1, -1), Point(1,␣
↪1, -1))), Point(0.0, 0.0, 0.0)), Pyramid(ConvexPolygen((Point(1, 1, 1), Point(-1,
↪ 1), Point(-1, -1, 1), Point(1, -1, 1))), Point(0.0, 0.0, 0.0))}
```

(continues on next page)

(continued from previous page)

```
point set:{Point(1, 1, -1), Point(-1, -1, -1), Point(1, -1, 1), Point(-1, 1, 1),␣
↪Point(1, 1, 1), Point(-1, -1, 1), Point(-1, 1, -1), Point(1, -1, -1)}
```

Specifically, Parallelepiped can be created using a Point and Three Vectors:

```
>>> cph = Parallelepiped(origin(),x_unit_vector(),y_unit_vector(),z_unit_vector())
>>> cph
ConvexPolyhedron
pyramid set:{Pyramid(ConvexPolygen((Point(1, 1, 1), Point(0, 1, 1), Point(0, 1, 0),␣
↪Point(1, 1, 0))), Point(0.5, 0.5, 0.5)), Pyramid(ConvexPolygen((Point(0, 0, 0),␣
↪Point(0, 1, 0), Point(0, 1, 1), Point(0, 0, 1))), Point(0.5, 0.5, 0.5)),␣
↪Pyramid(ConvexPolygen((Point(0, 0, 0), Point(1, 0, 0), Point(1, 0, 1), Point(0, 0,␣
↪1))), Point(0.5, 0.5, 0.5)), Pyramid(ConvexPolygen((Point(1, 1, 1), Point(1, 0, 1),␣
↪Point(1, 0, 0), Point(1, 1, 0))), Point(0.5, 0.5, 0.5)),␣
↪Pyramid(ConvexPolygen((Point(0, 0, 0), Point(1, 0, 0), Point(1, 1, 0), Point(0, 1,␣
↪0))), Point(0.5, 0.5, 0.5)), Pyramid(ConvexPolygen((Point(1, 1, 1), Point(0, 1, 1),␣
↪Point(0, 0, 1), Point(1, 0, 1))), Point(0.5, 0.5, 0.5))}
point set:{Point(0, 0, 1), Point(1, 1, 1), Point(1, 1, 0), Point(0, 1, 1), Point(1, 0,
↪ 1), Point(0, 0, 0), Point(1, 0, 0), Point(0, 1, 0)}
```

3.2 Renderer Examples

3.2.1 Creating Geometries

```
>>> a = Point(1,2,1)
>>> c = Point(-1,-1,1)
>>> d = Point(1,-1,1)
>>> e = Point(1,1,-1)
>>> h = Point(1,-1,-1)
>>>
>>> s = Segment(a,c)
>>>
>>> cpg = ConvexPolygen((a,d,h,e))
>>>
>>> cph = Parallelepiped(Point(-1.5,-1.5,-1.5),Vector(2,0,0),Vector(0,2,0),Vector(0,0,
↪2))
```

3.2.2 Getting a Renderer

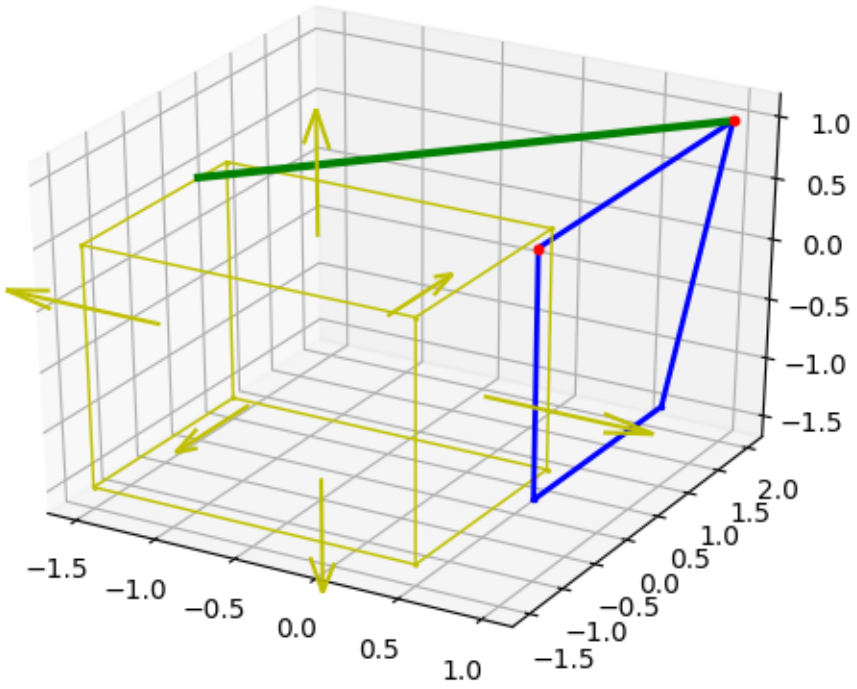
```
>>> r = Renderer(backend='matplotlib')
```

3.2.3 Adding Geometries

```
>>> r.add((a,'r',10),normal_length=0)
>>> r.add((d,'r',10),normal_length=0)
>>> r.add((s,'g',3),normal_length=0)
>>> r.add((cpg,'b',2),normal_length=0)
>>> r.add((cph,'y',1),normal_length=1)
```

3.2.4 Displaying Geometries

```
>>> r.show()
```



3.3 Getting Attributes

3.3.1 Creating Geometries

```
>>> a = Point(1,1,1)
>>> d = Point(1,-1,1)
>>> c = Point(-1,-1,1)
>>> e = Point(1,1,-1)
>>> h = Point(1,-1,-1)
>>>
>>> s = Segment(a,c)
>>>
>>> cpg = ConvexPolygen((a,d,h,e))
>>>
>>> cph = Parallelepiped(Point(-1,-1,-1),Vector(2,0,0),Vector(0,2,0),Vector(0,0,2))
```

3.3.2 Calculating the length

```
>>> s.length() # 2 * sqrt(2)
2.8284271247461903
>>> cpg.length() # 8
8.0
>>> cph.length() # 24
24.0
```

3.3.3 Calculating the area

```
>>> cph.area() # 24
23.999999999999993
>>> cpg.area() # 4
3.9999999999999982
>>> # Floating point calculation error
```

3.3.4 Calculating the volume

```
>>> cph.volume() # 8
7.999999999999995
>>> volume(cph0) # 8
7.999999999999995
```

3.4 Operations Examples

3.4.1 move

Move a Point:

```
>>> a = Point(1,2,1)
>>> print('a before move:{}'.format(a))
a before move:Point(1, 2, 1)
>>> a.move(x_unit_vector())
Point(2, 2, 1)
>>> print('a after move:{}'.format(a))
a after move:Point(2, 2, 1)
```

Move a Segment:

```
>>> b = origin()
>>> c = Point(1,2,3)
>>> s = Segment(b,c)
>>> s
Segment(Point(0, 0, 0), Point(1, 2, 3))
>>> s.move(Vector(-1,-2,-3))
Segment(Point(-1, -2, -3), Point(0, 0, 0))
>>> s
Segment(Point(-1, -2, -3), Point(0, 0, 0))
```

Move a ConvexPolygen **Without** Changing the Original Object:

```
>>> import copy
>>> cpg0 = Parallelogram(origin(),x_unit_vector(),y_unit_vector())
>>> cpg0
ConvexPolygen((Point(0, 0, 0), Point(1, 0, 0), Point(1, 1, 0), Point(0, 1, 0)))
>>> cpg1 = copy.deepcopy(cpg0).move(Vector(0,0,1))
>>> cpg0
ConvexPolygen((Point(0, 0, 0), Point(1, 0, 0), Point(1, 1, 0), Point(0, 1, 0)))
>>> cpg1
ConvexPolygen((Point(0, 0, 1), Point(1, 0, 1), Point(1, 1, 1), Point(0, 1, 1)))
```

3.4.2 Intersection

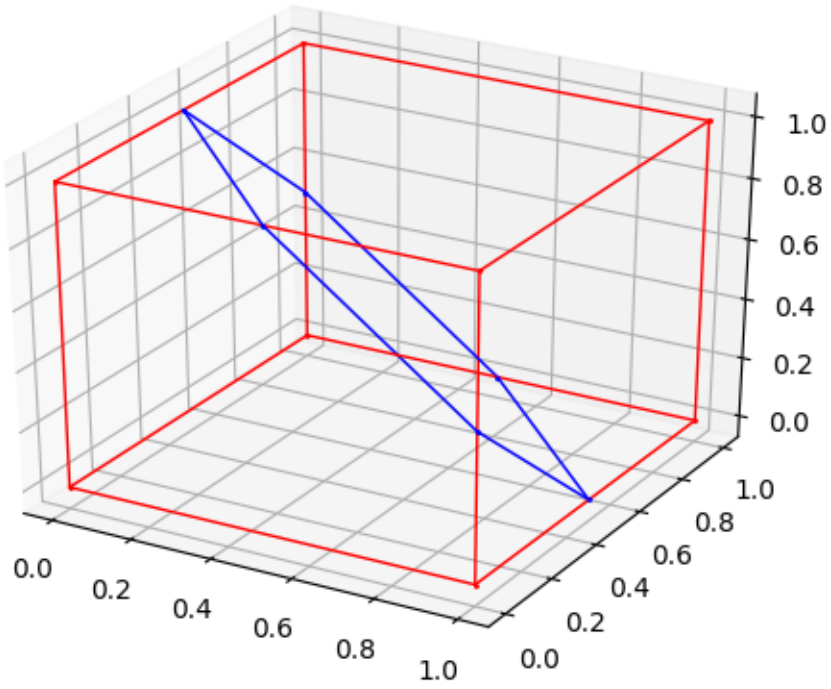
The operation of intersection is very complex. There are a total of 21 situations.

obj1	obj2	output obj
Point	Point	None, Point
Point	Line	None, Point
Point	Plane	None, Point
Point	Segment	None, Point
Point	ConvexPolygen	None, Point
Point	ConvexPolyhedron	None, Point
Line	Line	None, Point, Line
Line	Plane	None, Point, Line
Line	Segment	None, Point, Segment
Line	ConvexPolygen	None, Point, Segment
Line	ConvexPolyhedron	None, Point, Segment
Plane	Plane	None, Line, Plane
Plane	Segment	None, Point, Segment
Plane	ConvexPolygen	None, Point, Segment, ConvexPolygen
Plane	ConvexPolyhedron	None, Point, Segment, ConvexPolygen
Segment	Segment	None, Point, Segment
Segment	ConvexPolygen	None, Point, Segment
Segment	ConvexPolyhedron	None, Point, Segment
ConvexPolygen	ConvexPolygen	None, Point, Segment, ConvexPolygen
ConvexPolygen	ConvexPolyhedron	None, Point, Segment, ConvexPolygen
ConvexPolyhedron	ConvexPolyhedron	None, Point, Segment, ConvexPolygen, ConvexPolyhedron

All of the situations above are implemented. The documentation shows some examples.

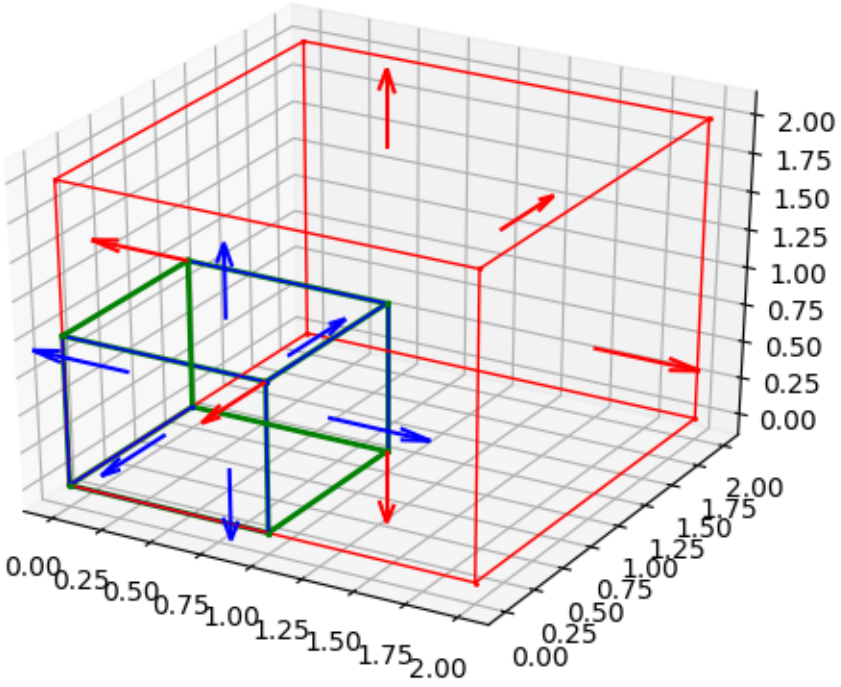
Example 1:

```
>>> po = origin()
>>> l1 = x_axis()
>>> l2 = y_axis()
>>> intersection(po,l1)
Point(0, 0, 0)
>>> intersection(l1,l2)
Point(0.0, 0.0, 0.0)
>>> s1 = Segment(Point(1,0,1),Point(0,1,1))
>>> s2 = Segment(Point(0,0,1),Point(1,1,1))
>>> s3 = Segment(Point(0.5,0.5,1),Point(-0.5,1.5,1))
>>> intersection(s1,s2)
Point(0.5, 0.5, 1.0)
>>> intersection(s1,s3)
Segment(Point(0.5, 0.5, 1.0), Point(0, 1, 1))
>>> intersection(l1,s1) is None
True
>>> cph0 = Parallelepiped(origin(),x_unit_vector(),y_unit_vector(),z_unit_vector())
>>> p = Plane(Point(0.5,0.5,0.5),Vector(1,1,1))
>>> cpg = intersection(cph0,p)
>>> r = Renderer()
>>> r.add((cph0,'r',1),normal_length = 0)
>>> r.add((cpg,'b',1),normal_length=0)
>>> r.show()
```



Example 2:

```
>>> from Geometry3D import *
>>> import copy
>>> r = Renderer()
>>> cph0 = Parallelepiped(origin(),x_unit_vector(),y_unit_vector(),z_unit_vector())
>>> cph6 = Parallelepiped(origin(),2 * x_unit_vector(),2 * y_unit_vector(),2 * z_unit_
↪vector())
>>> r.add((cph0,'b',1),normal_length = 0.5)
>>> r.add((cph6,'r',1),normal_length = 0.5)
>>> r.add((intersection(cph6,cph0),'g',2))
>>> print(intersection(cph0,cph6))
ConvexPolyhedron
pyramid set:{Pyramid(ConvexPolygen((Point(1, 1, 1), Point(0, 1, 1), Point(0.0, 0.0, 1.
↪0), Point(1, 0, 1))), Point(0.5, 0.5, 0.5)), Pyramid(ConvexPolygen((Point(1.0, 0.0,␣
↪0.0), Point(1, 0, 1), Point(1, 1, 1), Point(1, 1, 0))), Point(0.5, 0.5, 0.5)),␣
↪Pyramid(ConvexPolygen((Point(1, 1, 0), Point(1, 1, 1), Point(0, 1, 1), Point(0.0, 1.
↪0, 0.0))), Point(0.5, 0.5, 0.5)), Pyramid(ConvexPolygen((Point(0, 0, 1), Point(0, 0,
↪ 0), Point(0, 1, 0), Point(0, 1, 1))), Point(0.5, 0.5, 0.5)),␣
↪Pyramid(ConvexPolygen((Point(1, 0, 0), Point(1, 0, 1), Point(0, 0, 1), Point(0, 0,␣
↪0))), Point(0.5, 0.5, 0.5)), Pyramid(ConvexPolygen((Point(1, 1, 0), Point(1, 0, 0),␣
↪Point(0, 0, 0), Point(0, 1, 0))), Point(0.5, 0.5, 0.5))}
point set:{Point(1, 1, 0), Point(1, 1, 1), Point(0, 0, 1), Point(0, 1, 0), Point(0, 1,
↪ 1), Point(1.0, 0.0, 0.0), Point(0, 0, 0), Point(1, 0, 1)}
>>> r.show()
```



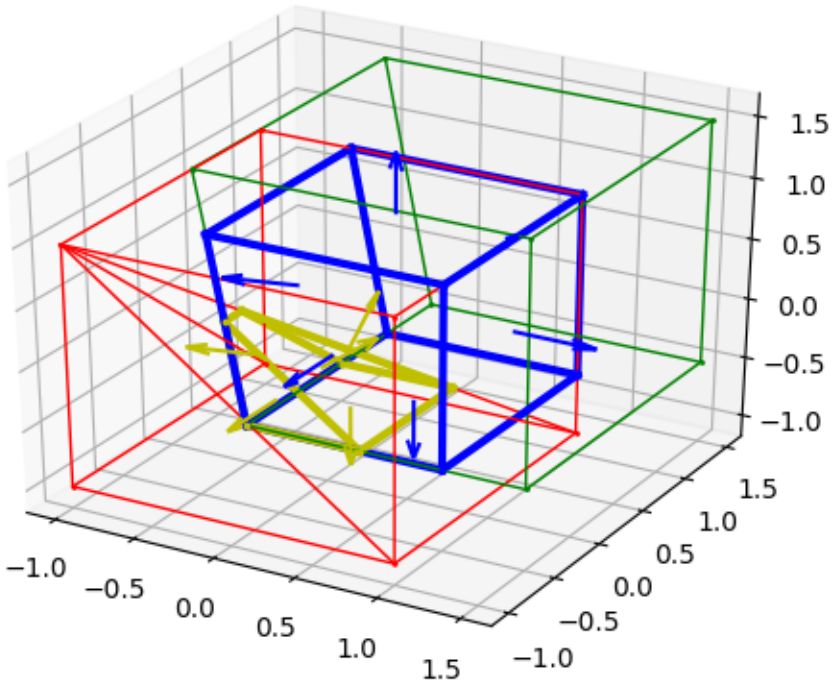
Example 3:

```
>>> from Geometry3D import *
>>>
>>> a = Point(1,1,1)
>>> b = Point(-1,1,1)
>>> c = Point(-1,-1,1)
>>> d = Point(1,-1,1)
>>> e = Point(1,1,-1)
>>> f = Point(-1,1,-1)
>>> g = Point(-1,-1,-1)
>>> h = Point(1,-1,-1)
>>> cph0 = Parallelepiped(Point(-1,-1,-1),Vector(2,0,0),Vector(0,2,0),Vector(0,0,2))
>>> cpg12 = ConvexPolygon((e,c,h))
>>> cpg13 = ConvexPolygon((e,f,c))
>>> cpg14 = ConvexPolygon((c,f,g))
>>> cpg15 = ConvexPolygon((h,c,g))
>>> cpg16 = ConvexPolygon((h,g,f,e))
>>> cph1 = ConvexPolyhedron((cpg12,cpg13,cpg14,cpg15,cpg16))
>>> a1 = Point(1.5,1.5,1.5)
>>> b1 = Point(-0.5,1.5,1.5)
>>> c1 = Point(-0.5,-0.5,1.5)
>>> d1 = Point(1.5,-0.5,1.5)
>>> e1 = Point(1.5,1.5,-0.5)
>>> f1 = Point(-0.2,1.5,-0.5)
>>> g1 = Point(-0.2,-0.5,-0.5)
```

(continues on next page)

(continued from previous page)

```
>>> h1 = Point(1.5,-0.5,-0.5)
>>>
>>> cpg6 = ConvexPolygen((a1,d1,h1,e1))
>>> cpg7 = ConvexPolygen((a1,e1,f1,b1))
>>> cpg8 = ConvexPolygen((c1,b1,f1,g1))
>>> cpg9 = ConvexPolygen((c1,g1,h1,d1))
>>> cpg10 = ConvexPolygen((a1,b1,c1,d1))
>>> cpg11 = ConvexPolygen((e1,h1,g1,f1))
>>> cph2 = ConvexPolyhedron((cpg6,cpg7,cpg8,cpg9,cpg10,cpg11))
>>> cph3 = intersection(cph0,cph2)
>>>
>>> cph4 = intersection(cph1,cph2)
>>> r = Renderer()
>>> r.add((cph0,'r',1),normal_length = 0)
>>> r.add((cph1,'r',1),normal_length = 0)
>>> r.add((cph2,'g',1),normal_length = 0)
>>> r.add((cph3,'b',3),normal_length = 0.5)
>>> r.add((cph4,'y',3),normal_length = 0.5)
>>> r.show()
```

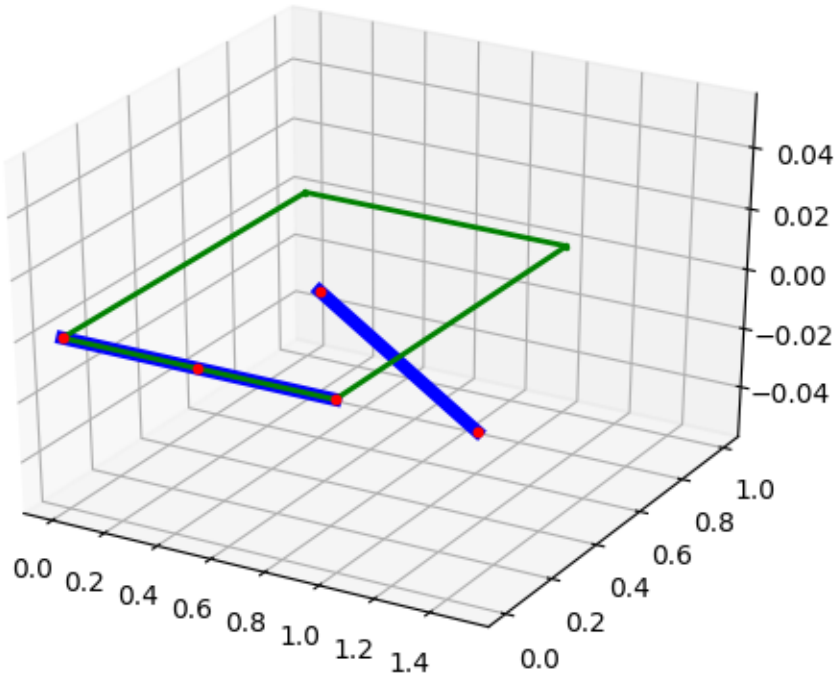


3.5 Build-In Functions

3.5.1 *__contains__*

__contains__ is used in build-in operator *in*, here are some examples:

```
>>> a = origin()
>>> b = Point(0.5,0,0)
>>> c = Point(1.5,0,0)
>>> d = Point(1,0,0)
>>> e = Point(0.5,0.5,0)
>>> s1 = Segment(origin(),d)
>>> s2 = Segment(e,c)
>>> a in s1
True
>>> b in s1
True
>>> c in s1
False
>>> a in s2
False
>>> b in s2
False
>>> c in s2
True
>>> cpg = Parallelogram(origin(),x_unit_vector(),y_unit_vector())
>>> a in cpg
True
>>> b in cpg
True
>>> c in cpg
False
>>> s1 in cpg
True
>>> s2 in cpg
False
>>>
>>> r=Renderer()
>>> r.add((a,'r',10))
>>> r.add((b,'r',10))
>>> r.add((c,'r',10))
>>> r.add((d,'r',10))
>>> r.add((e,'r',10))
>>> r.add((s1,'b',5))
>>> r.add((s2,'b',5))
>>> r.add((cpg,'g',2))
>>> r.show()
```



3.5.2 *__hash__*

__hash__ is used in set, here are some examples:

```
>>> a = set()
>>> a.add(origin())
>>> a
{Point(0, 0, 0)}
>>> a.add(Point(0,0,0))
>>> a
{Point(0, 0, 0)}
>>> a.add(Point(0,0,0.01))
>>> a
{Point(0, 0, 0), Point(0.0, 0.0, 0.01)}
>>>
>>> b = set()
>>> b.add(Segment(origin(),Point(1,0,0)))
>>> b
{Segment(Point(0, 0, 0), Point(1, 0, 0))}
>>> b.add(Segment(Point(1.0,0,0),Point(0,0,0)))
>>> b
{Segment(Point(0, 0, 0), Point(1, 0, 0))}
>>> b.add(Segment(Point(0,0,0),Point(0,1,1)))
>>> b
```

(continues on next page)

(continued from previous page)

```
{Segment(Point(0, 0, 0), Point(1, 0, 0)), Segment(Point(0, 0, 0), Point(0, 1, 1))}
```

3.5.3 *__eq__*

__eq__ is the build-in operator ==, here are some examples:

```
>>> a = origin()
>>> b = Point(1,0,0)
>>> c = Point(0,0,0)
>>> d = Point(2,0,0)
>>> a == b
False
>>> a == c
True
>>>
>>> s1 = Segment(a,b)
>>> s2 = Segment(a,b)
>>> s3 = Segment(b,a)
>>> s4 = Segment(a,d)
>>> s1 == s2
True
>>> s1 == s3
True
>>> s1 == s4
False
>>>
>>> cpg0 = ConvexPolygen((origin(),Point(1,0,0),Point(0,1,0),Point(1,1,0)))
>>> cpg1 = Parallelogram(origin(),x_unit_vector(),y_unit_vector())
>>> cpg0 == cpg1
True
```

3.5.4 *__neg__*

__neg__ is the build-in operator -, here are some examples:

```
>>> p = Plane(origin(),z_unit_vector())
>>> p
Plane(Point(0, 0, 0), Vector(0, 0, 1))
>>> -p
Plane(Point(0, 0, 0), Vector(0, 0, -1))
```

3.6 Dealing With Floating Numbers

There will be some errors in floating numbers computations. So identical objects may be deemed different. To tackle with this problem, this library believe two objects equal if their difference is smaller that a small number *eps*. Another value is named *significant number* has the relationship with eps:

```
significant number = -log(eps)
```

The default value of *eps* is 1e-10. You can access and change the value as follows:

```
>>> get_eps()
1e-10
>>> get_sig_figures()
10
>>> set_sig_figures(5)
>>> get_eps()
1e-05
>>> get_sig_figures()
5
>>> set_eps(1e-12)
>>> get_eps()
1e-12
>>> get_sig_figures()
12
```

CHAPTER

FOUR

PYTHON API

4.1 Geometry3D.calc package

4.1.1 Submodules

4.1.2 Geometry3D.calc.acute module

Acute Module

Geometry3D.calc.acute.**acute**(*rad*)

Input:

- rad: A angle in rad.

Output:

If the given angle is >90° (pi/2), return the opposite angle.

Return the angle else.

4.1.3 Geometry3D.calc.angle module

Angle Module

Geometry3D.calc.angle.**angle**(*a*, *b*)

Input:

- a: Line/Plane/Plane/Vector
- b: Line/Line/Plane/Vector

Output:

The angle (in radians) between

- Line/Line
- Plane/Line
- Plane/Plane
- Vector/Vector

Geometry3D.calc.angle.**parallel**(*a*, *b*)

Input:

- a:Line/Plane/Plane/Vector

- b:Line/Line/Plane/Vector

Output:

A boolean of whether the two objects are parallel. This can check

- Line/Line
- Plane/Line
- Plane/Plane
- Vector/Vector

Geometry3D.calc.angle.**orthogonal**(*a*, *b*)

Input:

- a:Line/Plane/Plane/Vector
- b:Line/Line/Plane/Vector

Output:

A boolean of whether the two objects are orthogonal. This can check

- Line/Line
- Plane/Line
- Plane/Plane
- Vector/Vector

4.1.4 Geometry3D.calc.aux_calc module

Auxilary Calculation Module.

Auxilary calculation functions for calculating intersection

Geometry3D.calc.aux_calc.**get_projection_length**(*v1*, *v2*)

Input:

- v1: Vector
- v2: Vector

Output:

The length of vector that v1 projected on v2

Geometry3D.calc.aux_calc.**get_relative_projection_length**(*v1*, *v2*)

Input:

- v1: Vector
- v2: Vector

Output:

The ratio of length of vector that v1 projected on v2 and the length of v2

Geometry3D.calc.aux_calc.**get_segment_from_point_list**(*point_list*)

Input:

- point_list: a list of Points

Output:

The longest segment between the points

Geometry3D.calc.aux_calc.**get_segment_convexpolyhedron_intersection_point_set**(*s*, *cph*)

Input:

- s: Segment
- cph: ConvexPolyhedron

Output:

A set of intersection points

Geometry3D.calc.aux_calc.**get_segment_convexpolygen_intersection_point_set**(*s*, *cpg*)

Input:

- s: Segment
- cpg: ConvexPolygen

Output:

A set of intersection points

Geometry3D.calc.aux_calc.**points_in_a_line**(*points*)

Input:

- points: Tuple or list of Points

Output:

A set of intersection points

4.1.5 Geometry3D.calc.distance module

Distance Module

Geometry3D.calc.distance.**distance**(*a*, *b*)

Input:

- a: Point/Line/Line/Plane/Plane
- b: Point/Point/Line/Point/Line

Output:

Returns the distance between two objects. This includes

- Point/Point
- Line/Point
- Line/Line
- Plane/Point
- Plane/Line

4.1.6 Geometry3D.calc.intersection module

Intersection Module

Geometry3D.calc.intersection.**intersection**(*a*, *b*)

Input:

- a: GeoBody or None
- b: GeoBody or None

Output:

The Intersection.

Maybe None or GeoBody

4.1.7 Geometry3D.calc.volume module

Volume module

Geometry3D.calc.volume.**volume**(*arg*)

Input:

- arg: Pyramid or ConvexPolyhedron

Output:

Returns the object volume. This includes

- Pyramid
- ConvexPolyhedron

4.1.8 Module contents

Geometry3D.calc.**distance**(*a*, *b*)

Input:

- a: Point/Line/Line/Plane/Plane
- b: Point/Point/Line/Point/Line

Output:

Returns the distance between two objects. This includes

- Point/Point
- Line/Point
- Line/Line
- Plane/Point
- Plane/Line

Geometry3D.calc.**intersection**(*a*, *b*)

Input:

- a: GeoBody or None
- b: GeoBody or None

Output:

The Intersection.

Maybe None or GeoBody

Geometry3D.calc.**parallel**(*a*, *b*)

Input:

- a:Line/Plane/Plane/Vector
- b:Line/Line/Plane/Vector

Output:

A boolean of whether the two objects are parallel. This can check

- Line/Line
- Plane/Line
- Plane/Plane
- Vector/Vector

Geometry3D.calc.**angle**(*a*, *b*)

Input:

- a: Line/Plane/Plane/Vector
- b: Line/Line/Plane/Vector

Output:

The angle (in radians) between

- Line/Line
- Plane/Line
- Plane/Plane
- Vector/Vector

Geometry3D.calc.**orthogonal**(*a*, *b*)

Input:

- a:Line/Plane/Plane/Vector
- b:Line/Line/Plane/Vector

Output:

A boolean of whether the two objects are orthogonal. This can check

- Line/Line
- Plane/Line
- Plane/Plane
- Vector/Vector

Geometry3D.calc.**volume**(*arg*)

Input:

- arg: Pyramid or ConvexPolyhedron

Output:

Returns the object volume. This includes

- Pyramid
- ConvexPolyhedron

Geometry3D.calc.**get_projection_length**(*v1*, *v2*)

Input:

- v1: Vector
- v2: Vector

Output:

The length of vector that v1 projected on v2

Geometry3D.calc.**get_relative_projection_length**(*v1*, *v2*)

Input:

- v1: Vector
- v2: Vector

Output:

The ratio of length of vector that v1 projected on v2 and the length of v2

Geometry3D.calc.**get_segment_from_point_list**(*point_list*)

Input:

- point_list: a list of Points

Output:

The longest segment between the points

Geometry3D.calc.**get_segment_convexpolyhedron_intersection_point_set**(*s*, *cph*)

Input:

- s: Segment
- cph: ConvexPolyhedron

Output:

A set of intersection points

Geometry3D.calc.**get_segment_convexpolygen_intersection_point_set**(*s*, *cpg*)

Input:

- s: Segment
- cpg: ConvexPolygen

Output:

A set of intersection points

Geometry3D.calc.**points_in_a_line**(*points*)

Input:

- points: Tuple or list of Points

Output:

A set of intersection points

4.2 Geometry3D.geometry package

4.2.1 Submodules

4.2.2 Geometry3D.geometry.body module

Geobody module

class Geometry3D.geometry.body.**GeoBody**
Bases: object

A base class for geometric objects that provides some common methods to work with. In the end, everything is dispatched to Geometry3D.calc.calc.* anyway, but it sometimes feels nicer to write it like L1.intersection(L2) instead of intersection(L1, L2)

angle(*other*)
return the angle between self and other

distance(*other*)
return the distance between self and other

intersection(*other*)
return the intersection between self and other

orthogonal(*other*)
return if self and other are orthogonal to each other

parallel(*other*)
return if self and other are parallel to each other

4.2.3 Geometry3D.geometry.line module

Line Module

class Geometry3D.geometry.line.**Line**(*a*, *b*)
Bases: *Geometry3D.geometry.body.GeoBody*

- Line(Point, Point):

A Line going through both given points.

- Line(Point, Vector):

A Line going through the given point, in the direction pointed by the given Vector.

- Line(Vector, Vector):

The same as Line(Point, Vector), but with instead of the point only the position vector of the point is given.

class_level = 1

move(*v*)
Return the line that you get when you move self by vector v, self is also moved

parametric()

Returns (s, u) so that you can build the equation for the line _ _ _

g: x = s + ru ; r e R

classmethod x_axis()
return x axis which is a Line

classmethod y_axis()
return y axis which is a Line

classmethod z_axis()
return z axis which is a Line

Geometry3D.geometry.line.**x_axis**()
return x axis which is a Line

Geometry3D.geometry.line.**y_axis**()
return y axis which is a Line

Geometry3D.geometry.line.**z_axis**()
return z axis which is a Line

4.2.4 Geometry3D.geometry.plane module

Plane module

class Geometry3D.geometry.plane.**Plane**(**args*)
Bases: *Geometry3D.geometry.body.GeoBody*

- Plane(Point, Point, Point):

Initialise a plane going through the three given points.

- Plane(Point, Vector, Vector):

Initialise a plane given by a point and two vectors lying on the plane.

- Plane(Point, Vector):

Initialise a plane given by a point and a normal vector (point normal form)

- Plane(a, b, c, d):

Initialise a plane given by the equation ax1 + bx2 + cx3 = d (general form).

class_level = 2

general_form()
Returns (a, b, c, d) so that you can build the equation

E: ax1 + bx2 + cx3 = d

to describe the plane.

move(*v*)
Return the plane that you get when you move self by vector v, self is also moved

parametric()

Returns (u, v, w) so that you can build the equation _ _ _ _

E: x = u + rv + sw ; (r, s) e R

to describe the plane (a point and two vectors).

point_normal()

Returns (p, n) so that you can build the equation _ _

E: (x - p) n = 0

to describe the plane.

classmethod xy_plane()
return xy plane which is a Plane

classmethod xz_plane()
return xz plane which is a Plane

classmethod yz_plane()
return yz plane which is a Plane

Geometry3D.geometry.plane.**xy_plane**()
return xy plane which is a Plane

Geometry3D.geometry.plane.**yz_plane**()
return yz plane which is a Plane

Geometry3D.geometry.plane.**xz_plane**()
return xz plane which is a Plane

4.2.5 Geometry3D.geometry.point module

Point Module

class Geometry3D.geometry.point.**Point**(**args*)
Bases: object

- Point(a, b, c)
- Point([a, b, c]):

The point with coordinates (a | b | c)

- Point(Vector):

The point that you get when you move the origin by the given vector. If the vector has coordinates (a | b | c), the point will have the coordinates (a | b | c) (as easy as pi).

class_level = 0

distance(*other*)
Return the distance between self and other

move(*v*)
Return the point that you get when you move self by vector v, self is also moved

classmethod origin()
Returns the Point (0 | 0 | 0)

pv()
Return the position vector of the point.

Geometry3D.geometry.point.**origin**()
Returns the Point (0 | 0 | 0)

4.2.6 Geometry3D.geometry.polygen module

Polygen Module

class Geometry3D.geometry.polygen.**ConvexPolygen**(*pts*, *reverse=False*, *check_convex=False*)

Bases: *Geometry3D.geometry.body.GeoBody*

- ConvexPolygens(points)

points: a tuple of points.

The points needn't to be in order.

The convexity should be guaranteed. This function **will not** check the convexity. If the Polygen is not convex, there might be errors.

classmethod Parallelogram(*base_point*, *v1*, *v2*)
A special function for creating Parallelogram

Input:

- base_point: a Point
- v1, v2: two Vectors

Output:

- A parallelogram which is a ConvexPolygen instance.

area()
Input:

- self

Output:

- The area of the convex polygen

class_level = 4

eq_with_normal(*other*)
return whether self equals with other considering the normal

hash_with_normal()
return the hash value considering the normal

in_(*other*)
Input:

- self: ConvexPolygen
- other: Plane

Output:

- whether self in other

length()
return the total length of ConvexPolygen

move(*v*)
Return the ConvexPolygen that you get when you move self by vector v, self is also moved

segments()
Input:

- self

Output:

- iterator of segments

Geometry3D.geometry.polygen.**Parallelogram**(*base_point*, *v1*, *v2*)
A special function for creating Parallelogram

Input:

- base_point: a Point
- v1, v2: two Vectors

Output:

- A parallelogram which is a ConvexPolygen instance.

4.2.7 Geometry3D.geometry.polyhedron module

Polyhedron Module

class Geometry3D.geometry.polyhedron.**ConvexPolyhedron**(*convex_polygens*)
Bases: *Geometry3D.geometry.body.GeoBody*

classmethod Parallelepiped(*base_point*, *v1*, *v2*, *v3*)
A special function for creating Parallelepiped

Input:

- base_point: a Point
- v1, v2, v3: three Vectors

Output:

- A parallelepiped which is a ConvexPolyhedron instance.

area()
return the total area of the polyhedron

class_level = 5
Input:

- convex_polygens: tuple of ConvexPolygens

Output:

- ConvexPolyhedron
- The correctness of convex_polygens are checked According to Euler's formula.
- The normal of the convex polygens are checked and corrected which should be toward the outer direction

length()
return the total length of the polyhedron

move(*v*)
Return the ConvexPolyhedron that you get when you move self by vector v, self is also moved

volume()
return the total volume of the polyhedron

Geometry3D.geometry.polyhedron.**Parallelepiped**(*base_point*, *v1*, *v2*, *v3*)
A special function for creating Parallelepiped

Input:

- base_point: a Point
- v1, v2, v3: three Vectors

Output:

- A parallelepiped which is a ConvexPolyhedron instance.

4.2.8 Geometry3D.geometry.pyramid module

Pyramid Module

class Geometry3D.geometry.pyramid.**Pyramid**(*cp*, *p*, *direct_call=True*)
Bases: *Geometry3D.geometry.body.GeoBody*

Input:

- cp: a ConvexPolygen
- p: a Point

height()
return the height of the pyramid

volume()
return the volume of the pryamid

4.2.9 Geometry3D.geometry.segment module

Segment Module

class Geometry3D.geometry.segment.**Segment**(*a*, *b*)
Bases: *Geometry3D.geometry.body.GeoBody*

Input:

- Segment(Point,Point)
- Segment(Point,Vector)

class_level = 3

in_(*other*)
other can be plane or line

length()
retutn the length of the segment

move(*v*)
Return the Segment that you get when you move self by vector v, self is also moved

parametric()
Returns (start_point, end_point) so that you can build the information for the segment

4.2.10 Module contents

class Geometry3D.geometry.**ConvexPolyhedron**(*convex_polygens*)
Bases: *Geometry3D.geometry.body.GeoBody*

classmethod Parallelepiped(*base_point*, *v1*, *v2*, *v3*)
A special function for creating Parallelepiped

Input:

- base_point: a Point
- v1, v2, v3: three Vectors

Output:

- A parallelepiped which is a ConvexPolyhedron instance.

area()
return the total area of the polyhedron

class_level = 5
Input:

- convex_polygens: tuple of ConvexPolygens

Output:

- ConvexPolyhedron
- The correctness of convex_polygens are checked According to Euler's formula.
- The normal of the convex polygens are checked and corrected which should be toward the outer direction

length()
return the total length of the polyhedron

move(*v*)
Return the ConvexPolyhedron that you get when you move self by vector v, self is also moved

volume()
return the total volume of the polyhedron

Geometry3D.geometry.**Parallelepiped**(*base_point*, *v1*, *v2*, *v3*)
A special function for creating Parallelepiped

Input:

- base_point: a Point
- v1, v2, v3: three Vectors

Output:

- A parallelepiped which is a ConvexPolyhedron instance.

class Geometry3D.geometry.**ConvexPolygen**(*pts*, *reverse=False*, *check_convex=False*)
Bases: *Geometry3D.geometry.body.GeoBody*

- ConvexPolygens(points)

points: a tuple of points.

The points needn't to be in order.

The convexity should be guaranteed. This function **will not** check the convexity. If the Polygen is not convex, there might be errors.

classmethod Parallelogram(*base_point*, *v1*, *v2*)
A special function for creating Parallelogram

Input:

- base_point: a Point
- v1, v2: two Vectors

Output:

- A parallelogram which is a ConvexPolygen instance.

area()
Input:

- self

Output:

- The area of the convex polygen

class_level = 4

eq_with_normal(*other*)
return whether self equals with other considering the normal

hash_with_normal()
return the hash value considering the normal

in_(*other*)
Input:

- self: ConvexPolygen
- other: Plane

Output:

- whether self in other

length()
return the total length of ConvexPolygen

move(*v*)
Return the ConvexPolygen that you get when you move self by vector v, self is also moved

segments()
Input:

- self

Output:

- iterator of segments

Geometry3D.geometry.**Parallelogram**(*base_point*, *v1*, *v2*)
A special function for creating Parallelogram

Input:

- base_point: a Point
- v1, v2: two Vectors

Output:

- A parallelogram which is a ConvexPolygen instance.

class Geometry3D.geometry.**Pyramid**(*cp*, *p*, *direct_call=True*)
Bases: *Geometry3D.geometry.body.GeoBody*

Input:

- cp: a ConvexPolygen
- p: a Point

height()
return the height of the pyramid

volume()
return the volume of the pryamid

class Geometry3D.geometry.**Segment**(*a*, *b*)
Bases: *Geometry3D.geometry.body.GeoBody*

Input:

- Segment(Point,Point)
- Segment(Point,Vector)

class_level = 3

in_(*other*)
other can be plane or line

length()
retutn the length of the segment

move(*v*)
Return the Segment that you get when you move self by vector v, self is also moved

parametric()
Returns (start_point, end_point) so that you can build the information for the segment

class Geometry3D.geometry.**Line**(*a*, *b*)
Bases: *Geometry3D.geometry.body.GeoBody*

- Line(Point, Point):

A Line going through both given points.

- Line(Point, Vector):

A Line going through the given point, in the direction pointed by the given Vector.

- Line(Vector, Vector):

The same as Line(Point, Vector), but with instead of the point only the position vector of the point is given.

class_level = 1

move(*v*)
Return the line that you get when you move self by vector v, self is also moved

parametric()

Returns (s, u) so that you can build the equation for the line _ _ _

g: x = s + ru ; r e R

classmethod x_axis()
return x axis which is a Line

classmethod y_axis()
return y axis which is a Line

classmethod z_axis()
return z axis which is a Line

class Geometry3D.geometry.**Plane**(**args*)
Bases: *Geometry3D.geometry.body.GeoBody*

- Plane(Point, Point, Point):

Initialise a plane going through the three given points.

- Plane(Point, Vector, Vector):

Initialise a plane given by a point and two vectors lying on the plane.

- Plane(Point, Vector):

Initialise a plane given by a point and a normal vector (point normal form)

- Plane(a, b, c, d):

Initialise a plane given by the equation ax1 + bx2 + cx3 = d (general form).

class_level = 2

general_form()
Returns (a, b, c, d) so that you can build the equation

E: ax1 + bx2 + cx3 = d

to describe the plane.

move(*v*)
Return the plane that you get when you move self by vector v, self is also moved

parametric()

Returns (u, v, w) so that you can build the equation _ _ _ _

E: x = u + rv + sw ; (r, s) e R

to describe the plane (a point and two vectors).

point_normal()

Returns (p, n) so that you can build the equation _ _

E: (x - p) n = 0

to describe the plane.

classmethod xy_plane()
return xy plane which is a Plane

classmethod xz_plane()
return xz plane which is a Plane

classmethod yz_plane()
return yz plane which is a Plane

class Geometry3D.geometry.**Point**(**args*)
Bases: object

- Point(a, b, c)
- Point([a, b, c]):

The point with coordinates (a | b | c)

- Point(Vector):

The point that you get when you move the origin by the given vector. If the vector has coordinates (a | b | c), the point will have the coordinates (a | b | c) (as easy as pi).

class_level = 0

distance(*other*)
Return the distance between self and other

move(*v*)
Return the point that you get when you move self by vector v, self is also moved

classmethod origin()
Returns the Point (0 | 0 | 0)

pv()
Return the position vector of the point.

Geometry3D.geometry.**origin**()
Returns the Point (0 | 0 | 0)

Geometry3D.geometry.**x_axis**()
return x axis which is a Line

Geometry3D.geometry.**y_axis**()
return y axis which is a Line

Geometry3D.geometry.**z_axis**()
return z axis which is a Line

Geometry3D.geometry.**xy_plane**()
return xy plane which is a Plane

Geometry3D.geometry.**yz_plane**()
return yz plane which is a Plane

Geometry3D.geometry.**xz_plane**()
return xz plane which is a Plane

4.3 Geometry3D.render package

4.3.1 Submodules

4.3.2 Geometry3D.render.arrow module

Arrow Module for Renderer

class Geometry3D.render.arrow.**Arrow**(*x*, *y*, *z*, *u*, *v*, *w*, *length*)
Bases: object

Arrow Class

get_tuple()
return the tuple expression of the arrow

4.3.3 Geometry3D.render.renderer module

Abstract Renderer Module

Geometry3D.render.renderer.**Renderer**(*backend='matplotlib'*)

Input:

- backend: the backend of the renderer

Only matplotlib is supported till now

4.3.4 Geometry3D.render.renderer_matplotlib module

Matplotlib Renderer Module

class Geometry3D.render.renderer_matplotlib.**MatplotlibRenderer**

Bases: object

class Axes3D(*fig*, *rect=None*, **args*, *azim=- 60*, *elev=30*, *zscale=None*, *sharez=None*, *proj_type='persp'*, ***kwargs*)

Bases: matplotlib.axes._axes.Axes

3D axes object.

add_collection3d(*col*, *zs=0*, *zdir='z'*)

Add a 3D collection object to the plot.

2D collection types are converted to a 3D version by modifying the object and adding z coordinate information.

Supported are:

- PolyCollection
- LineCollection
- PatchCollection

add_contour_set(*cset*, *extend3d=False*, *stride=5*, *zdir='z'*, *offset=None*)

add_contourf_set(*cset*, *zdir='z'*, *offset=None*)

auto_scale_xyz(*X*, *Y*, *Z=None*, *had_data=None*)

autoscale(*enable=True*, *axis='both'*, *tight=None*)

Convenience method for simple axis view autoscaling. See matplotlib.axes.Axes.autoscale() for full explanation. Note that this function behaves the same, but for all three axes. Therefore, 'z' can be passed for *axis*, and 'both' applies to all three axes.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

autoscale_view(*tight=None*, *scalex=True*, *scaley=True*, *scalez=True*)

Autoscale the view limits using the data limits. See matplotlib.axes.Axes.autoscale_view() for documentation. Note that this function applies to the 3D axes, and as such adds the *scalez* to the function arguments.

Changed in version 1.1.0: Function signature was changed to better match the 2D version. *tight* is now explicitly a kwarg and placed first.

Changed in version 1.2.1: This is now fully functional.

bar(*left*, *height*, *zs=0*, *zdir='z'*, **args*, ***kwargs*)

Add 2D bar(s).

Argument	Description
left	The x coordinates of the left sides of the bars.
height	The height of the bars.
zs	Z coordinate of bars, if one value is specified they will all be placed at the same z.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.

Keyword arguments are passed onto `bar()`.

Returns a `Patch3DCollection`

bar3d(*x*, *y*, *z*, *dx*, *dy*, *dz*, *color=None*, *zsort='average'*, *shade=True*, **args*, ***kwargs*)
Generate a 3D barplot.

This method creates three dimensional barplot where the width, depth, height, and color of the bars can all be uniquely set.

x, y, z [array-like] The coordinates of the anchor point of the bars.

dx, dy, dz [scalar or array-like] The width, depth, and height of the bars, respectively.

color [sequence of valid color specifications, optional] The color of the bars can be specified globally or individually. This parameter can be:

- A single color value, to color all bars the same color.
- An array of colors of length N bars, to color each bar independently.
- An array of colors of length 6, to color the faces of the bars similarly.
- An array of colors of length 6 * N bars, to color each face independently.

When coloring the faces of the boxes specifically, this is the order of the coloring:

1. -Z (bottom of box)
2. +Z (top of box)
3. -Y
4. +Y
5. -X
6. +X

zsort [str, optional] The z-axis sorting scheme passed onto `Poly3DCollection()`

shade [bool, optional (default = True)] When true, this shades the dark sides of the bars (relative to the plot's source of light).

Any additional keyword arguments are passed onto `Poly3DCollection()`

collection [Poly3DCollection] A collection of three dimensional polygons representing the bars.

can_pan()
Return *True* if this axes supports the pan/zoom button functionality.

3D axes objects do not use the pan/zoom button.

can_zoom()
Return *True* if this axes supports the zoom box button functionality.

3D axes objects do not use the zoom box button.

cla()
Clear axes

clabel(**args*, ***kwargs*)
This function is currently not implemented for 3D axes. Returns *None*.

contour(*X*, *Y*, *Z*, **args*, *extend3d=False*, *stride=5*, *zdir='z'*, *offset=None*, ***kwargs*)
Create a 3D contour plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
extend3d	Whether to extend contour in 3D (default: False)
stride	Stride (step size) for extending contour
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the contour lines on this position in plane normal to zdir

The positional and other keyword arguments are passed on to `contour()`

Returns a `contour`

contour3D(*X*, *Y*, *Z*, **args*, *extend3d=False*, *stride=5*, *zdir='z'*, *offset=None*, ***kwargs*)
Create a 3D contour plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
extend3d	Whether to extend contour in 3D (default: False)
stride	Stride (step size) for extending contour
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the contour lines on this position in plane normal to zdir

The positional and other keyword arguments are passed on to `contour()`

Returns a `contour`

contourf(*X*, *Y*, *Z*, **args*, *zdir='z'*, *offset=None*, ***kwargs*)
Create a 3D contourf plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the filled contour on this position in plane normal to zdir

The positional and keyword arguments are passed on to `contourf()`

Returns a `contourf`

Changed in version 1.1.0: The *zdir* and *offset* kwargs were added.

contourf3D(*X*, *Y*, *Z*, **args*, *zdir='z'*, *offset=None*, ***kwargs*)
Create a 3D contourf plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the filled contour on this position in plane normal to zdir

The positional and keyword arguments are passed on to `contourf()`

Returns a `contourf`

Changed in version 1.1.0: The *zdir* and *offset* kwargs were added.

convert_zunits(*z*)
For artists in an axes, if the zaxis has units support, convert *z* using zaxis unit type

New in version 1.2.1.

disable_mouse_rotation()
Disable mouse button callbacks.

draw(*renderer*)
Draw everything (plot lines, axes, labels)

format_coord(*xd*, *yd*)
Given the 2D view coordinates attempt to guess a 3D coordinate. Looks for the nearest edge to the point and then assumes that the point is at the same z location as the nearest point on the edge.

format_zdata(*z*)
Return *z* string formatted. This function will use the `fmt_zdata` attribute if it is callable, else will fall back on the zaxis major formatter

get_autoscale_on()
Get whether autoscaling is applied for all axes on plot commands

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

get_autoscalez_on()
Get whether autoscaling for the z-axis is applied on plot commands

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

get_axis_position()

get_children()
return a list of child artists

get_frame_on()
Get whether the 3D axes panels are drawn.

New in version 1.1.0.

get_proj()
Create the projection matrix from the current viewing position.

elev stores the elevation angle in the z plane azim stores the azimuth angle in the x,y plane

dist is the distance of the eye viewing point from the object point.

get_w_lims()
Get 3D world limits.

get_xlim()
Return the x-axis view limits.
left, right [(float, float)] The current x-axis limits in data coordinates.
set_xlim set_xbound, get_xbound invert_xaxis, xaxis_inverted

The x-axis may be inverted, in which case the *left* value will be greater than the *right* value.
Changed in version 1.1.0: This function now correctly refers to the 3D x-limits

get_xlim3d()
Return the x-axis view limits.
left, right [(float, float)] The current x-axis limits in data coordinates.
set_xlim set_xbound, get_xbound invert_xaxis, xaxis_inverted

The x-axis may be inverted, in which case the *left* value will be greater than the *right* value.
Changed in version 1.1.0: This function now correctly refers to the 3D x-limits

get_ylim()
Return the y-axis view limits.
bottom, top [(float, float)] The current y-axis limits in data coordinates.
set_ylim set_ybound, get_ybound invert_yaxis, yaxis_inverted

The y-axis may be inverted, in which case the *bottom* value will be greater than the *top* value.
Changed in version 1.1.0: This function now correctly refers to the 3D y-limits.

get_ylim3d()
Return the y-axis view limits.
bottom, top [(float, float)] The current y-axis limits in data coordinates.
set_ylim set_ybound, get_ybound invert_yaxis, yaxis_inverted

The y-axis may be inverted, in which case the *bottom* value will be greater than the *top* value.
Changed in version 1.1.0: This function now correctly refers to the 3D y-limits.

get_zbound()
Returns the z-axis numerical bounds where:

```
lowerBound < upperBound
```

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

get_zlabel()
Get the z-label text string.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

get_zlim()
Get 3D z limits.

get_zlim3d()
Get 3D z limits.

get_zmajorticklabels()
Get the ztick labels as a list of Text instances

New in version 1.1.0.

get_zminorticklabels()
Get the ztick labels as a list of Text instances

Note: Minor ticks are not supported. This function was added only for completeness.

New in version 1.1.0.

get_zscale()

get_zticklabels(*minor=False*)
Get ztick labels as a list of Text instances. See `matplotlib.axes.Axes.get_yticklabels()` for more details.

Note: Minor ticks are not supported.

New in version 1.1.0.

get_zticklines()
Get ztick lines as a list of Line2D instances. Note that this function is provided merely for completeness. These lines are re-calculated as the display changes.

New in version 1.1.0.

get_zticks(*minor=False*)
Return the z ticks as a list of locations See `matplotlib.axes.Axes.get_yticks()` for more details.

Note: Minor ticks are not supported.

New in version 1.1.0.

grid(*b=True*, ***kwargs*)
Set / unset 3D grid.

Note: Currently, this function does not behave the same as `matplotlib.axes.Axes.grid()`, but it is intended to eventually support that behavior.

Changed in version 1.1.0: This function was changed, but not tested. Please report any bugs.

have_units()
Return *True* if units are set on the *x*, *y*, or *z* axes

invert_zaxis()
Invert the z-axis.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

locator_params(*axis='both'*, *tight=None*, ***kwargs*)
Convenience method for controlling tick locators.

See `matplotlib.axes.Axes.locator_params()` for full documentation. Note that this is for Axes3D objects, therefore, setting *axis* to 'both' will result in the parameters being set for all three axes. Also, *axis* can also take a value of 'z' to apply parameters to the z axis.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

margins(**margins*, *x=None*, *y=None*, *z=None*, *tight=True*)
Convenience method to set or retrieve autoscaling margins.
signatures:: margins()
returns xmargin, ymargin, zmargin

```
margins(margin)

margins(xmargin, ymargin, zmargin)
```

(continues on next page)

(continued from previous page)

```
margins(x=xmargin, y=ymargin, z=zmargin)

margins(..., tight=False)
```

All forms above set the xmargin, ymargin and zmargin parameters. All keyword parameters are optional. A single positional argument specifies xmargin, ymargin and zmargin. Passing both positional and keyword arguments for xmargin, ymargin, and/or zmargin is invalid.

The *tight* parameter is passed to `autoscale_view()`, which is executed after a margin is changed; the default here is *True*, on the assumption that when margins are specified, no additional padding to match tick marks is usually desired. Setting *tight* to *None* will preserve the previous setting.

Specifying any margin changes only the autoscaling; for example, if *xmargin* is not None, then *xmargin* times the X data interval will be added to each end of that interval before it is used in autoscaling.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

mouse_init(*rotate_btn=1*, *zoom_btn=3*)

Initializes mouse button callbacks to enable 3D rotation of the axes. Also optionally sets the mouse buttons for 3D rotation and zooming.

Argument	Description
rotate_btn	The integer or list of integers specifying which mouse button or buttons to use for 3D rotation of the axes. Default = 1.
zoom_btn	The integer or list of integers specifying which mouse button or buttons to use to zoom the 3D axes. Default = 3.

name = '3d'

plot(*xs*, *ys*, **args*, *zdir='z'*, ***kwargs*)

Plot 2D or 3D data.

Argument	Description
xs, *ys*	x, y coordinates of vertices
zs	z value(s), either one for all points or one for each point.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.

Other arguments are passed on to `plot()`

plot3D(*xs*, *ys*, **args*, *zdir='z'*, ***kwargs*)

Plot 2D or 3D data.

Argument	Description
xs, *ys*	x, y coordinates of vertices
zs	z value(s), either one for all points or one for each point.
zdir	Which direction to use as z ('x', 'y' or 'z') when plotting a 2D set.

Other arguments are passed on to `plot()`

plot_surface(*X*, *Y*, *Z*, **args*, *norm=None*, *vmin=None*, *vmax=None*, *lightsource=None*, ***kwargs*)

Create a surface plot.

By default it will be colored in shades of a solid color, but it also supports color mapping by supplying the *cmap* argument.

Note: The *rcount* and *ccount* kwargs, which both default to 50, determine the maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points.

X, Y, Z [2d arrays] Data values.
rcount, ccount [int] Maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points. Defaults to 50.

New in version 2.0.
rstride, cstride [int] Downsampling stride in each direction. These arguments are mutually exclusive with *rcount* and *ccount*. If only one of *rstride* or *cstride* is set, the other defaults to 10.

'classic' mode uses a default of `rstride = cstride = 10` instead of the new default of `rcount = ccount = 50`.
color [color-like] Color of the surface patches.
cmap [Colormap] Colormap of the surface patches.
facecolors [array-like of colors.] Colors of each individual patch.
norm [Normalize] Normalization for the colormap.
vmin, vmax [float] Bounds for the normalization.
shade [bool] Whether to shade the face colors.
****kwargs :** Other arguments are forwarded to *.Poly3DCollection*.

plot_trisurf(**args*, *color=None*, *norm=None*, *vmin=None*, *vmax=None*, *lightsource=None*, ***kwargs*)

Argument	Description
X, *Y*, *Z*	Data values as 1D arrays
color	Color of the surface patches
cmap	A colormap for the surface patches.
norm	An instance of Normalize to map values to colors
vmin	Minimum value to map
vmax	Maximum value to map
shade	Whether to shade the facecolors

The (optional) triangulation can be specified in one of two ways; either:

```
plot_trisurf(triangulation, ...)
```

where triangulation is a `Triangulation` object, or:

```
plot_trisurf(X, Y, ...)
plot_trisurf(X, Y, triangles, ...)
plot_trisurf(X, Y, triangles=triangles, ...)
```

in which case a Triangulation object will be created. See `Triangulation` for a explanation of these possibilities.

The remaining arguments are:

```
plot_trisurf(..., Z)
```

where *Z* is the array of values to contour, one per point in the triangulation.

Other arguments are passed on to `Poly3DCollection`

Examples:

New in version 1.2.0: This plotting function was added for the v1.2.0 release.

plot_wireframe (*X*, *Y*, *Z*, **args*, ***kwargs*)
Plot a 3D wireframe.

Note: The *rcount* and *ccount* kwargs, which both default to 50, determine the maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points.

X, Y, Z [2d arrays] Data values.

rcount, ccount [int] Maximum number of samples used in each direction. If the input data is larger, it will be downsampled (by slicing) to these numbers of points. Setting a count to zero causes the data to be not sampled in the corresponding direction, producing a 3D line plot rather than a wireframe plot. Defaults to 50.

New in version 2.0.

rstride, cstride [int] Downsampling stride in each direction. These arguments are mutually exclusive with *rcount* and *ccount*. If only one of *rstride* or *cstride* is set, the other defaults to 1. Setting a stride to zero causes the data to be not sampled in the corresponding direction, producing a 3D line plot rather than a wireframe plot.

'classic' mode uses a default of `rstride = cstride = 1` instead of the new default of `rcount = ccount = 50`.

****kwargs :** Other arguments are forwarded to *.Line3DCollection*.

quiver (**args*, *length=1*, *arrow_length_ratio=0.3*, *pivot='tail'*, *normalize=False*, ***kwargs*)
Plot a 3D field of arrows.

call signatures:

```
quiver(X, Y, Z, U, V, W, **kwargs)
```

Arguments:

***X, Y, Z*:** The x, y and z coordinates of the arrow locations (default is tail of arrow; see *pivot* kwarg)

***U, V, W*:** The x, y and z components of the arrow vectors

The arguments could be array-like or scalars, so long as they they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

Keyword arguments:

***length*: [1.0 | float]** The length of each quiver, default to 1.0, the unit is the same with the axes

***arrow_length_ratio*: [0.3 | float]** The ratio of the arrow head with respect to the quiver, default to 0.3

***pivot*: ['tail' | 'middle' | 'tip']** The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name *pivot*. Default is 'tail'

***normalize*: bool** When True, all of the arrows will be the same length. This defaults to False, where the arrows will be different lengths depending on the values of u,v,w.

Any additional keyword arguments are delegated to `LineCollection`

quiver3D (**args*, *length=1*, *arrow_length_ratio=0.3*, *pivot='tail'*, *normalize=False*, ***kwargs*)
Plot a 3D field of arrows.

call signatures:

```
quiver(X, Y, Z, U, V, W, **kwargs)
```

Arguments:

***X*, *Y*, *Z*:** The x, y and z coordinates of the arrow locations (default is tail of arrow; see *pivot* kwarg)

***U*, *V*, *W*:** The x, y and z components of the arrow vectors

The arguments could be array-like or scalars, so long as they they can be broadcast together. The arguments can also be masked arrays. If an element in any of argument is masked, then that corresponding quiver element will not be plotted.

Keyword arguments:

***length*: [1.0 | float]** The length of each quiver, default to 1.0, the unit is the same with the axes

***arrow_length_ratio*: [0.3 | float]** The ratio of the arrow head with respect to the quiver, default to 0.3

***pivot*: ['tail' | 'middle' | 'tip']** The part of the arrow that is at the grid point; the arrow rotates about this point, hence the name *pivot*. Default is 'tail'

***normalize*: bool** When True, all of the arrows will be the same length. This defaults to False, where the arrows will be different lengths depending on the values of u,v,w.

Any additional keyword arguments are delegated to `LineCollection`

`scatter` (*xs*, *ys*, *zs=0*, *zdir='z'*, *s=20*, *c=None*, *depthshade=True*, **args*, ***kwargs*)

Create a scatter plot.

xs, ys [array-like] The data positions.

zs [float or array-like, optional, default: 0] The z-positions. Either an array of the same length as *xs* and *ys* or a single value to place all points in the same plane.

zdir [{'x', 'y', 'z', '-x', '-y', '-z'}, optional, default: 'z'] The axis direction for the *zs*. This is useful when plotting 2D data on a 3D Axes. The data must be passed as *xs*, *ys*. Setting *zdir* to 'y' then plots the data to the x-z-plane.

See also /gallery/mplot3d/2dcollections3d.

s [scalar or array-like, optional, default: 20] The marker size in points**2. Either an array of the same length as *xs* and *ys* or a single value to make all markers the same size.

c [color, sequence, or sequence of color, optional] The marker color. Possible values:

- A single color format string.
- A sequence of color specifications of length n.
- A sequence of n numbers to be mapped to colors using *cmap* and *norm*.
- A 2-D array in which the rows are RGB or RGBA.

For more details see the *c* argument of *~.axes.Axes.scatter*.

depthshade [bool, optional, default: True] Whether to shade the scatter markers to give the appearance of depth.

****kwargs** All other arguments are passed on to *~.axes.Axes.scatter*.

paths : *~matplotlib.collections.PathCollection*

`scatter3D` (*xs*, *ys*, *zs=0*, *zdir='z'*, *s=20*, *c=None*, *depthshade=True*, **args*, ***kwargs*)

Create a scatter plot.

xs, ys [array-like] The data positions.

zs [float or array-like, optional, default: 0] The z-positions. Either an array of the same length as *xs* and *ys* or a single value to place all points in the same plane.

zdir [{'x', 'y', 'z', '-x', '-y', '-z'}, optional, default: 'z'] The axis direction for the *zs*. This is useful when plotting 2D data on a 3D Axes. The data must be passed as *xs*, *ys*. Setting *zdir* to 'y' then plots the data to the x-z-plane.

See also /gallery/mplot3d/2dcollections3d.

s [scalar or array-like, optional, default: 20] The marker size in points**2. Either an array of the same length as *xs* and *ys* or a single value to make all markers the same size.

c [color, sequence, or sequence of color, optional] The marker color. Possible values:

- A single color format string.
- A sequence of color specifications of length n.
- A sequence of n numbers to be mapped to colors using *cmap* and *norm*.
- A 2-D array in which the rows are RGB or RGBA.

For more details see the *c* argument of *~.axes.Axes.scatter*.

depthshade [bool, optional, default: True] Whether to shade the scatter markers to give the appearance of depth.

****kwargs** All other arguments are passed on to *~.axes.Axes.scatter*.

paths : *~matplotlib.collections.PathCollection*

set_autoscale_on(*b*)

Set whether autoscaling is applied on plot commands

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

b : bool

set_autoscalez_on(*b*)

Set whether autoscaling for the z-axis is applied on plot commands

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

b : bool

set_axis_off()

Turn the x- and y-axis off.

This affects the axis lines, ticks, ticklabels, grid and axis labels.

set_axis_on()

Turn the x- and y-axis on.

This affects the axis lines, ticks, ticklabels, grid and axis labels.

set_frame_on(*b*)

Set whether the 3D axes panels are drawn.

New in version 1.1.0.

b : bool

set_proj_type(*proj_type*)

Set the projection type.

proj_type [str] Type of projection, accepts 'persp' and 'ortho'.

set_title(*label*, *fontdict=None*, *loc='center'*, ***kwargs*)

Set a title for the axes.

Set one of the three available axes titles. The available titles are positioned above the axes in the center, flush with the left edge, and flush with the right edge.

label [str] Text to use for the title

fontdict [dict] A dictionary controlling the appearance of the title text, the default *fontdict* is:

```
{'fontsize': rcParams['axes.titlesize'],
 'fontweight' : rcParams['axes.titleweight'],
 'verticalalignment': 'baseline',
 'horizontalalignment': loc}
```

loc [{'center', 'left', 'right'}, str, optional] Which title to set, defaults to 'center'

pad [float] The offset of the title from the top of the axes, in points. Default is `None` to use rcParams['axes.titlepad'].

text [`Text`] The matplotlib text instance representing the title

****kwargs** [*~matplotlib.text.Text* properties] Other keyword arguments are text properties, see `Text` for a list of valid text properties.

set_top_view()

set_xlim(*left=None*, *right=None*, *emit=True*, *auto=False*, ***, *xmin=None*, *xmax=None*)
Set 3D x limits.

See `matplotlib.axes.Axes.set_xlim()` for full documentation.

set_xlim3d(*left=None*, *right=None*, *emit=True*, *auto=False*, ***, *xmin=None*, *xmax=None*)
Set 3D x limits.

See `matplotlib.axes.Axes.set_xlim()` for full documentation.

set_xscale(*value*, ***kwargs*)
Set the x-axis scale.
value [{"linear", "log", "symlog", "logit", ... }] The axis scale type to apply.
****kwargs** Different keyword arguments are accepted, depending on the scale. See the respective class keyword arguments:
- *matplotlib.scale.LinearScale*
- *matplotlib.scale.LogScale*
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.scale.LogitScale*

By default, Matplotlib supports the above mentioned scales. Additionally, custom scales may be registered using *matplotlib.scale.register_scale*. These scales can then also be used here.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_ylim(*bottom=None*, *top=None*, *emit=True*, *auto=False*, ***, *ymin=None*, *ymax=None*)
Set 3D y limits.

See `matplotlib.axes.Axes.set_ylim()` for full documentation.

set_ylim3d(*bottom=None*, *top=None*, *emit=True*, *auto=False*, ***, *ymin=None*, *ymax=None*)
Set 3D y limits.

See `matplotlib.axes.Axes.set_ylim()` for full documentation.

set_yscale(*value*, ***kwargs*)
Set the y-axis scale.
value [{"linear", "log", "symlog", "logit", ... }] The axis scale type to apply.
****kwargs** Different keyword arguments are accepted, depending on the scale. See the respective class keyword arguments:
- *matplotlib.scale.LinearScale*
- *matplotlib.scale.LogScale*
- *matplotlib.scale.SymmetricalLogScale*
- *matplotlib.scale.LogitScale*

By default, Matplotlib supports the above mentioned scales. Additionally, custom scales may be registered using *matplotlib.scale.register_scale*. These scales can then also be used here.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_zbound(*lower=None*, *upper=None*)
Set the lower and upper numerical bounds of the z-axis. This method will honor axes inversion regardless of parameter order. It will not change the `_autoscaleZon` attribute.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_zlabel(*zlabel*, *fontdict=None*, *labelpad=None*, ***kwargs*)
Set zlabel. See doc for `set_ylabel()` for description.

set_zlim(*bottom=None*, *top=None*, *emit=True*, *auto=False*, ***, *zmin=None*, *zmax=None*)
Set 3D z limits.

See `matplotlib.axes.Axes.set_ylim()` for full documentation

set_zlim3d(*bottom=None*, *top=None*, *emit=True*, *auto=False*, ***, *zmin=None*, *zmax=None*)
Set 3D z limits.

See `matplotlib.axes.Axes.set_ylim()` for full documentation

set_zmargin(*m*)
Set padding of Z data limits prior to autoscaling.

m times the data interval will be added to each end of that interval before it is used in autoscaling.

accepts: float in range 0 to 1

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_zscale(*value*, ***kwargs*)
Set the scaling of the z-axis: 'linear' | 'log' | 'logit' | 'symlog'

ACCEPTS: ['linear' | 'log' | 'logit' | 'symlog']
Different kwargs are accepted, depending on the scale: 'linear'

'log'
***basex/basey*:** The base of the logarithm
***nonposx/nonposy*: {'mask', 'clip'}** non-positive values in *x* or *y* can be masked as invalid, or clipped to a very small positive number
***subsx/subsy*:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

'logit'
***nonpos*: {'mask', 'clip'}** values beyond]0, 1[can be masked as invalid, or clipped to a number very close to 0 or 1

'symlog'
***basex/basey*:** The base of the logarithm
***linthreshx/linthreshy*:** A single float which defines the range (*-x*, *x*), within which the plot is linear. This avoids having the plot go to infinity around zero.
***subsx/subsy*:** Where to place the subticks between each major tick. Should be a sequence of integers. For example, in a log10 scale: `[2, 3, 4, 5, 6, 7, 8, 9]`

will place 8 logarithmically spaced minor ticks between each major tick.

***linscalex/linscaley*:** This allows the linear range (*-linthresh* to *linthresh*) to be stretched relative to the logarithmic range. Its value is the number of decades to use for each half of the linear range. For example, when *linscale* == 1.0 (the default), the space used for the positive and negative halves of the linear range will be equal to one decade in the logarithmic range.

Note: Currently, Axes3D objects only supports linear scales. Other scales may or may not work, and support for these is improving with each release.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

set_zticklabels(**args*, ***kwargs*)
Set z-axis tick labels. See `matplotlib.axes.Axes.set_yticklabels()` for more details.

Note: Minor ticks are not supported by Axes3D objects.

New in version 1.1.0.

set_zticks (**args*, ***kwargs*)
Set z-axis tick locations. See `matplotlib.axes.Axes.set_yticks()` for more details.

Note: Minor ticks are not supported.

New in version 1.1.0.

text (*x*, *y*, *z*, *s*, *zdir=None*, ***kwargs*)
Add text to the plot. kwargs will be passed on to Axes.text, except for the *zdir* keyword, which sets the direction to be used as the z direction.

text2D (*x*, *y*, *s*, *fontdict=None*, *withdash=False*, ***kwargs*)
Add text to the axes.

Add the text *s* to the axes at location *x*, *y* in data coordinates.

x, y [scalars] The position to place the text. By default, this is in data coordinates. The coordinate system can be changed using the *transform* parameter.

s [str] The text.

fontdict [dictionary, optional, default: None] A dictionary to override the default text properties. If fontdict is None, the defaults are determined by your rc parameters.

withdash [boolean, optional, default: False] Creates a *~matplotlib.text.TextWithDash* instance instead of a *~matplotlib.text.Text* instance.

text [*.Text*] The created *.Text* instance.

****kwargs** [*~matplotlib.text.Text* properties.] Other miscellaneous text parameters.

Individual keyword arguments can be used to override any given parameter:

```
>>> text(x, y, s, fontsize=12)
```

The default transform specifies that text is in data coords, alternatively, you can specify text in axis coords (0,0 is lower-left and 1,1 is upper-right). The example below places text in the center of the axes:

```
>>> text(0.5, 0.5, 'matplotlib', horizontalalignment='center',
...      verticalalignment='center', transform=ax.transAxes)
```

You can put a rectangular box around the text instance (e.g., to set a background color) by using the keyword *bbox*. *bbox* is a dictionary of *~matplotlib.patches.Rectangle* properties. For example:

```
>>> text(x, y, s, bbox=dict(facecolor='red', alpha=0.5))
```

text3D (*x*, *y*, *z*, *s*, *zdir=None*, ***kwargs*)
Add text to the plot. kwargs will be passed on to Axes.text, except for the *zdir* keyword, which sets the direction to be used as the z direction.

tick_params (*axis='both'*, ***kwargs*)
Convenience method for changing the appearance of ticks and tick labels.

See `matplotlib.axes.Axes.tick_params()` for more complete documentation.

The only difference is that setting *axis* to 'both' will mean that the settings are applied to all three axes. Also, the *axis* parameter also accepts a value of 'z', which would mean to apply to only the z-axis.

Also, because of how Axes3D objects are drawn very differently from regular 2D axes, some of these settings may have ambiguous meaning. For simplicity, the 'z' axis will accept settings as if it was like the 'y' axis.

Note: While this function is currently implemented, the core part of the Axes3D object may ignore some of these settings. Future releases will fix this. Priority will be given to those who file bugs.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

ticklabel_format(*, *style=''*, *scilimits=None*, *useOffset=None*, *axis='both'*)

Convenience method for manipulating the ScalarFormatter used by default for linear axes in Axed3D objects.

See `matplotlib.axes.Axes.ticklabel_format()` for full documentation. Note that this version applies to all three axes of the Axes3D object. Therefore, the *axis* argument will also accept a value of 'z' and the value of 'both' will apply to all three axes.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

tricontour(**args*, *extend3d=False*, *stride=5*, *zdir='z'*, *offset=None*, ***kwargs*)

Create a 3D contour plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
extend3d	Whether to extend contour in 3D (default: False)
stride	Stride (step size) for extending contour
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the contour lines on this position in plane normal to zdir

Other keyword arguments are passed on to `tricontour()`

Returns a `contour`

Changed in version 1.3.0: Added support for custom triangulations

EXPERIMENTAL: This method currently produces incorrect output due to a longstanding bug in 3D PolyCollection rendering.

tricontourf(**args*, *zdir='z'*, *offset=None*, ***kwargs*)

Create a 3D contourf plot.

Argument	Description
X, *Y*,	Data values as numpy.arrays
Z	
zdir	The direction to use: x, y or z (default)
offset	If specified plot a projection of the contour lines on this position in plane normal to zdir

Other keyword arguments are passed on to `tricontour()`

Returns a `contour`

Changed in version 1.3.0: Added support for custom triangulations

EXPERIMENTAL: This method currently produces incorrect output due to a longstanding bug in 3D PolyCollection rendering.

tunit_cube(*vals=None*, *M=None*)

tunit_edges(*vals=None*, *M=None*)

unit_cube(*vals=None*)

update_datalim(*xys*, ***kwargs*)
Extend the ~*.Axes.dataLim* BBox to include the given points.

If no data is set currently, the BBox will ignore its limits and set the bound to be the bounds of the xydata (*xys*). Otherwise, it will compute the bounds of the union of its current data and the data in *xys*.

xys [2D array-like] The points to include in the data limits BBox. This can be either a list of (x, y) tuples or a Nx2 array.

updatex, updatey [bool, optional, default *True*] Whether to update the x/y limits.

view_init(*elev=None*, *azim=None*)
Set the elevation and azimuth of the axes.

This can be used to rotate the axes programmatically.

'elev' stores the elevation angle in the z plane. 'azim' stores the azimuth angle in the x,y plane.

if elev or azim are None (default), then the initial value is used which was specified in the `Axes3D` constructor.

voxels(*[x, y, z], /, filled*, ***kwargs*)
Plot a set of filled voxels

All voxels are plotted as 1x1x1 cubes on the axis, with filled[0,0,0] placed with its lower corner at the origin. Occluded faces are not plotted.

Call signatures:

```
voxels(filled, facecolors=fc, edgecolors=ec, **kwargs)
voxels(x, y, z, filled, facecolors=fc, edgecolors=ec, **kwargs)
```

New in version 2.1.

filled [3D np.array of bool] A 3d array of values, with truthy values indicating which voxels to fill

x, y, z [3D np.array, optional] The coordinates of the corners of the voxels. This should broadcast to a shape one larger in every dimension than the shape of *filled*. These can be used to plot non-cubic voxels.

If not specified, defaults to increasing integers along each axis, like those returned by `indices()`. As indicated by the `/` in the function signature, these arguments can only be passed positionally.

facecolors, edgecolors [array_like, optional] The color to draw the faces and edges of the voxels. Can only be passed as keyword arguments. This parameter can be:

- A single color value, to color all voxels the same color. This can be either a string, or a 1D rgb/rgba array
- `None`, the default, to use a single color for the faces, and the style default for the edges.
- A 3D ndarray of color names, with each item the color for the corresponding voxel. The size must match the voxels.

- A 4D ndarray of rgb/rgba data, with the components along the last axis.

****kwargs** Additional keyword arguments to pass onto `Poly3DCollection()`

faces [dict] A dictionary indexed by coordinate, where `faces[i,j,k]` is a *Poly3DCollection* of the faces drawn for the voxel `filled[i,j,k]`. If no faces were drawn for a given voxel, either because it was not asked to be drawn, or it is fully occluded, then `(i,j,k) not in faces`.

zaxis_date(*tz=None*)

Sets up z-axis ticks and labels that treat the z data as dates.

tz is a timezone string or `tzinfo` instance. Defaults to rc value.

Note: This function is merely provided for completeness. Axes3D objects do not officially support dates for ticks, and so this may or may not work as expected.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

zaxis_inverted()

Returns True if the z-axis is inverted.

New in version 1.1.0: This function was added, but not tested. Please report any bugs.

add(*obj*, *normal_length=0*)

Input:

- obj: a tuple (object,color,size)
- normal_length: the length of normal arrows for ConvexPolyhedron.

For other objects, normal_length should be zero. If you don't want to show the normal arrows for a ConvexPolyhedron, you can set normal_length to 0.

object can be Point, Segment, ConvexPolygen or ConvexPolyhedron

mpl = <module 'matplotlib' from '/home/gmh/pyenv/s3d/lib/python3.5/site-packages/matpl

plt = <module 'matplotlib.pyplot' from '/home/gmh/pyenv/s3d/lib/python3.5/site-package

show()

Draw the image

4.3.5 Module contents

`Geometry3D.render.`**Renderer**(*backend='matplotlib'*)

Input:

- backend: the backend of the renderer

Only matplotlib is supported till now

4.4 Geometry3D.utils package

4.4.1 Submodules

4.4.2 Geometry3D.utils.constant module

Constant module

EPS and significant numbers for comparing float point numbers.

Two float numbers are deemed equal if they equal with each other within significant numbers.

Significant numbers = log(1 / eps) all the time

`Geometry3D.utils.constant.`**`set_eps`**(*eps=1e-10*)

Input:

- eps: floating number with 1e-10 the default

Output:

No output but set EPS to eps

Signigicant numbers is also changed.

`Geometry3D.utils.constant.`**`get_eps`**()

Input:

no input

Output:

- current eps: float

`Geometry3D.utils.constant.`**`get_sig_figures`**()

Input:

no input

Output:

- current significant numbers: int

`Geometry3D.utils.constant.`**`set_sig_figures`**(*sig_figures=10*)

Input:

- sig_figures: int with 10 the default

Output:

No output but set signigicant numbers to sig_figures

EPS is also changed.

4.4.3 Geometry3D.utils.logger module

Logger Module

Geometry3D.utils.logger.**change_main_logger**()

Geometry3D.utils.logger.**get_main_logger**()

Input:

No Input

Output:

main_logger: The logger instance

Geometry3D.utils.logger.**set_log_level**(*level='WARNING'*)

Input:

- level: a string of log level among 'DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL'.

  'WARNING' is the default.

Output:

No output but setup the log level for the logger

4.4.4 Geometry3D.utils.solver module

Solver Module, An Auxilary Module

class Geometry3D.utils.solver.**Solution**(*s*)

Bases: object

Holds a solution to a system of equations.

Geometry3D.utils.solver.**count**(*f*, *l*)

Geometry3D.utils.solver.**find_pivot_row**(*m*)

Geometry3D.utils.solver.**first_nonzero**(*r*)

Geometry3D.utils.solver.**gaussian_elimination**(*m*)

Return the row echelon form of m by applying the gaussian elimination

Geometry3D.utils.solver.**index**(*f*, *l*)

Geometry3D.utils.solver.**null**(*f*)

Geometry3D.utils.solver.**nullrow**(*r*)

Geometry3D.utils.solver.**shape**(*m*)

Geometry3D.utils.solver.**solve**(*matrix*)

4.4.5 Geometry3D.utils.util module

Util Module

Geometry3D.utils.util.**unify_types**(*items*)

Promote all items to the same type. The resulting type is the "most valueable" that an item already has as defined by the list (top = least valueable):

- int
- float
- decimal.Decimal
- fractions.Fraction
- user defined

4.4.6 Geometry3D.utils.vector module

Vector Module

class Geometry3D.utils.vector.**Vector**(**args*)

Bases: object

Vector Class

angle(*other*)

Returns the angle (in radians) enclosed by both vectors.

cross(*other*)

Calculates the cross product of two vectors, defined as _ _ / x2y3 - x3y2 x × y = | x3y1 - x1y3 |

x1y2 - x2y1 /

The cross product is orthogonal to both vectors and its length is the area of the parallelogram given by x and y.

length()

Returns |v|, the length of the vector.

normalized()

Return the normalized version of the vector, that is a vector pointing in the same direction but with length 1.

orthogonal(*other*)

Returns true if the two vectors are orthogonal

parallel(*other*)

Returns true if both vectors are parallel.

unit()

Return the normalized version of the vector, that is a vector pointing in the same direction but with length 1.

classmethod x_unit_vector()

Returns the unit vector (1 | 0 | 0)

classmethod y_unit_vector()

Returns the unit vector (0 | 1 | 0)

classmethod z_unit_vector()

Returns the unit vector (0 | 0 | 1)

classmethod zero()
Returns the zero vector (0 | 0 | 0)

Geometry3D.utils.vector.**x_unit_vector**()
Returns the unit vector (1 | 0 | 0)

Geometry3D.utils.vector.**y_unit_vector**()
Returns the unit vector (0 | 1 | 0)

Geometry3D.utils.vector.**z_unit_vector**()
Returns the unit vector (0 | 0 | 1)

4.4.7 Module contents

Geometry3D.utils.**solve**(*matrix*)

class Geometry3D.utils.**Vector**(**args*)
Bases: object

Vector Class

angle(*other*)
Returns the angle (in radians) enclosed by both vectors.

cross(*other*)
Calculates the cross product of two vectors, defined as _ _ / x2y3 - x3y2 x × y = | x3y1 - x1y3 |

x1y2 - x2y1 /

The cross product is orthogonal to both vectors and its length is the area of the parallelogram given by x and y.

length()
Returns |v|, the length of the vector.

normalized()
Return the normalized version of the vector, that is a vector pointing in the same direction but with length 1.

orthogonal(*other*)
Returns true if the two vectors are orthogonal

parallel(*other*)
Returns true if both vectors are parallel.

unit()
Return the normalized version of the vector, that is a vector pointing in the same direction but with length 1.

classmethod x_unit_vector()
Returns the unit vector (1 | 0 | 0)

classmethod y_unit_vector()
Returns the unit vector (0 | 1 | 0)

classmethod z_unit_vector()
Returns the unit vector (0 | 0 | 1)

classmethod zero()
Returns the zero vector (0 | 0 | 0)

Geometry3D.utils.**x_unit_vector**()
Returns the unit vector (1 | 0 | 0)

Geometry3D.utils.**y_unit_vector**()
Returns the unit vector (0 | 1 | 0)

Geometry3D.utils.**z_unit_vector**()
Returns the unit vector (0 | 0 | 1)

Geometry3D.utils.**set_eps**(*eps=1e-10*)
Input:

- eps: floating number with 1e-10 the default

Output:

No output but set EPS to eps

Signigicant numbers is also changed.

Geometry3D.utils.**get_eps**()
Input:

no input

Output:

- current eps: float

Geometry3D.utils.**get_sig_figures**()
Input:

no input

Output:

- current significant numbers: int

Geometry3D.utils.**set_sig_figures**(*sig_figures=10*)
Input:

- sig_figures: int with 10 the default

Output:

No output but set signigicant numbers to sig_figures

EPS is also changed.

Geometry3D.utils.**set_log_level**(*level='WARNING'*)
Input:

- level: a string of log level among 'DEBUG', 'INFO', 'WARNING', 'ERROR', 'CRITICAL'.

  'WARNING' is the default.

Output:

No output but setup the log level for the logger

Geometry3D.utils.**get_main_logger**()
Input:

No Input

Output:

main_logger: The logger instance

CHAPTER

FIVE

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

INDEX

D

E

F

G

N

O

P

Q

R

S

T

U

V

X

Y

Z